From Computing to Intelligence

A First-Principles Approach to Artificial Intelligence in Language, Vision, and Audio

Sagar Udasi

MSc Statistics with Data Science and Computational Finance
University of Edinburgh

From Computing to Intelligence

A First-Principles Approach to Artificial Intelligence in Language, Vision, and Audio

Copyright © 2025 by Sagar Udasi

All rights reserved.
No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher or author.

ISBN: 9798285270225

9 798285 270225

Author: Sagar Udasi
Publisher: Amazon Kindle Direct Publishing
First Edition

Printed in the United Kingdom

For information, please contact:
Sagar Udasi
Email: `sagar.l.udasi@gmail.com`

Contents

Introduction

Who Is This Book For?

What This Book Covers and What It Doesn’t

Welcome to **From Computing to Intelligence**.

Sagar Udasi

SECTION 1

FOUNDATIONS OF INTELLIGENCE

Chapter 1

Neural Networks

Humans have always considered themselves intelligent, perhaps too intelligent. For centuries, philosophers, anatomists, and scientists stared into the mirror of the mind, trying to grasp what makes intelligence possible. Naturally, the question arose: *How do we think?* And if we could understand that, *could we build a machine that thinks?* To build an artificial intelligence, we need to take inspiration from nature's most optimized real intelligent machine — the brain!

1.1 The Neuron

In 1873, an Italian scientist named ***Camillo Golgi*** discovered a silver staining technique that revealed something extraordinary: the fine cellular architecture of the brain. In 1890, a Spanish neuroscientist and artist, ***Santiago Ramón y Cajal*** took Golgi's stain and produced breathtaking drawings that showed brain as connected architecture of individual, separate units — not a continuous mesh as many thought. Cajal proposed what we now call the *Neuron Doctrine*: that the brain is made up of individual units, now known as **neurons**, that communicate across small gaps by transmitting chemicals (signals), which we now call **synapses**.

The exact working of the brain was little understood by the 1940s. However, one thing was known: neurons form a network, and based on the signals they receive from other neurons, they decide whether to release a signal themselves, a process called *firing*, or not. Enter ***Warren McCulloch***, a neurophysiologist, and ***Walter Pitts***, a brilliant young logician who had taught himself formal logic as a teenager. In 1943, in a small research lab in Chicago, they published a revolutionary paper: *A Logical Calculus of the Ideas Immanent in Nervous Activity* [1].

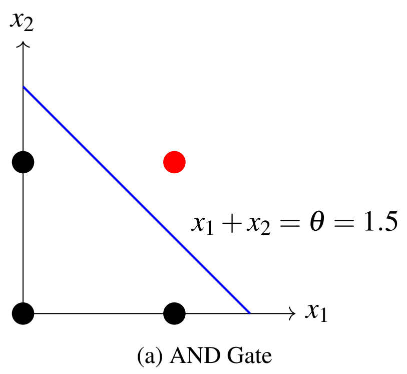
In their paper they proposed that each neuron functions as a binary threshold unit. Neurons receive multiple input signals, aggregate them, and produce an output (i.e., *fire*) only if the aggregated signal crosses a certain threshold. Formally, the **McCulloch-Pitts model** defines a neuron that receives inputs $\mathbf{x} = (x_1,\ x_2,\ \ldots,\ x_n)$, where any input $x_i \in \{0, 1\}$. The neuron aggregates the inputs by computing the sum $g(\mathbf{x}) = \sum_{i=1}^{n} x_i$.

The output is then determined by a fixed constant known as the *thresholding parameter*, θ, as

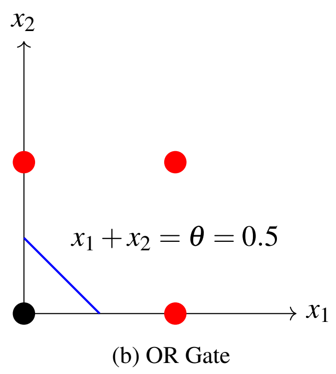
$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \theta \\ 0 & \text{if } g(\mathbf{x}) < \theta \end{cases}$$

Geometrically, a single McCulloch-Pitts neuron divides the space of input points into two regions by a decision boundary defined by the equation $\sum_{i=1}^{n} x_i - \theta = 0$. For the case of two binary inputs, this line partitions the four possible input points into two groups: those for which $\sum x_i < \theta$ (which produce output 0) and those for which $\sum x_i \geq \theta$ (which produce output 1). Thus, the decision boundary acts as a separator in the input space.

This idea naturally extends to higher dimensions. If the neuron receives three inputs, the line becomes a plane: $\sum_{i=1}^{3} x_i - \theta = 0$. For instance, in the case of the 3-input OR function, the desired plane must separate the point (0,0,0) which should produce output 0, from the other seven binary combinations, which should all produce output 1.



(a) AND Gate



(b) OR Gate

Figure 1.1: McCulloch-Pitts neuron implementing logical functions via thresholding

The McCulloch-Pitts neuron provides a foundational abstraction of a biological neuron, but it also raises several important questions. *What if the inputs are not binary but real-valued? Do we always need to manually specify the threshold parameter? Are all inputs equally important, or should some be assigned more weight?* Finally, *can such a neuron model functions that are not linearly separable?*

1.2 The Perceptron

To address these concerns, ***Frank Rosenblatt*** introduced the *Perceptron* in 1958, funded by the U.S. Navy. Rosenblatt's model generalized the McCulloch-Pitts neuron by allowing real-valued inputs, learnable weights, and thresholds.

Formally, the perceptron computes a weighted sum of its inputs and applies a threshold to determine the binary output.

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i - \theta \geq 0 \\ 0 & \text{if } \sum_{i=1}^{n} w_i x_i - \theta < 0 \end{cases}$$

A more commonly used convention rewrites this using a *bias* term.

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i \geq 0 \\ 0 & \text{if } \sum_{i=0}^{n} w_i x_i < 0 \end{cases} \quad \text{where } x_0 = 1 \text{ and } w_0 = -\theta$$

This formulation shows that a perceptron still divides the input space into two halves. Inputs that produce an output of 1 lie on one side of the decision boundary $\sum w_i x_i = 0$, while those producing 0 lie on the other.

In other words, a single perceptron can only model linearly separable functions. However, it differs from the McCulloch-Pitts model in two significant ways: the weights (including the threshold) are learnable from data, and the inputs can be real-valued.

Let us revisit the Boolean OR function. For inputs $x_1, x_2 \in \{0,1\}$, we desire an output $y = x_1 \vee x_2$. The perceptron should satisfy the following inequalities.

$\mathbf{x_1}$	$\mathbf{x_2}$	**OR**	**Inequality**	**Condition**
0	0	0	$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0$	$\Rightarrow w_0 < 0$
1	0	1	$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0$	$\Rightarrow w_1 \geq -w_0$
0	1	1	$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0$	$\Rightarrow w_2 \geq -w_0$
1	1	1	$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0$	$\Rightarrow w_1 + w_2 \geq -w_0$

Table 1.1: Linear inequalities that represent the OR gate using weights w_0, w_1, and w_2

One possible solution satisfying all the above constraints is $w_0 = -1$, $w_1 = 1.1$, and $w_2 = 1.1$.

Note that this is not a unique solution—many combinations of weights satisfy the inequalities. The key insight is that for linearly separable functions like OR, a perceptron can find suitable weights to model the function accurately. We will later explore how these weights can be learned automatically using the *perceptron learning algorithm*.

1.2.1 Perceptron Learning Algorithm

Imagine we want to make a binary decision of whether to watch a movie or not. Suppose we are given a list of m movies, each labeled as either liked (1) or not liked (0) by a user. Each movie is represented by n features, which can be either Boolean or real-valued. We assume the data is linearly separable and aim to train a perceptron to learn this classification rule. In other words, we want the perceptron to find the parameters $\mathbf{w} = [w_0, w_1, \ldots, w_n]$ that define a separating hyperplane.

We now describe the perceptron learning algorithm formally.

Perceptron Learning Algorithm

```
Let $P \leftarrow$ set of inputs labeled 1
Let $N \leftarrow$ set of inputs labeled 0
Initialize weight vector $\mathbf{w}$ randomly
while not converged do
    Pick a random example $\mathbf{x} \in P \cup N$
    if $\mathbf{x} \in P$ and $\mathbf{w}^T\mathbf{x} < 0$ then
        $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$
    if $\mathbf{x} \in N$ and $\mathbf{w}^T\mathbf{x} \geq 0$ then
        $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}$
end while
```

The algorithm continues updating the weights until all inputs are correctly classified. *But why does this work?*

To understand this, consider two vectors: the weight vector $\mathbf{w} = [w_0, w_1, \ldots, w_n]$ and an input vector $\mathbf{x} = [1, x_1, \ldots, x_n]$. The perceptron computes the dot product

$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^{n} w_i x_i$$

The classification rule can then be written as

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} \geq 0 \\ 0 & \text{if } \mathbf{w}^T \mathbf{x} < 0 \end{cases}$$

Geometrically, the equation $\mathbf{w}^T \mathbf{x} = 0$ defines a hyperplane that splits the input space into two halves. Any point $\mathbf{x}$ lying on this hyperplane satisfies $\mathbf{w}^T \mathbf{x} = 0$. The weight vector $\mathbf{w}$ is orthogonal to this hyperplane, because the angle α between $\mathbf{w}$ and any vector $\mathbf{x}$ on the hyperplane is $90°$, which implies $\cos\alpha = 0$ and hence $\mathbf{w}^T \mathbf{x} = 0$.

Now consider a point $\mathbf{x} \in P$ (positive class) such that $\mathbf{w}^T \mathbf{x} < 0$. This implies that the angle between $\mathbf{w}$ and $\mathbf{x}$ is greater than $90°$, i.e., $\mathbf{x}$ lies in the wrong half-space. We wish to reduce this angle so that $\mathbf{x}$ lies in the correct half-space. Updating $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$ has the following effect

$$\mathbf{w}_{\text{new}}^T \mathbf{x} = (\mathbf{w} + \mathbf{x})^T \mathbf{x} = \mathbf{w}^T \mathbf{x} + \mathbf{x}^T \mathbf{x}$$

Since $\mathbf{x}^T \mathbf{x} > 0$, we have $\mathbf{w}_{\text{new}}^T \mathbf{x} > \mathbf{w}^T \mathbf{x}$, which implies $\cos(\alpha_{\text{new}}) > \cos(\alpha)$ and therefore $\alpha_{\text{new}} < \alpha$. This update moves the decision boundary in the desired direction, reducing misclassification.

Similarly, for a point $\mathbf{x} \in N$ (negative class) such that $\mathbf{w}^T \mathbf{x} \geq 0$, the vector $\mathbf{x}$ lies in the wrong half-space. The angle between $\mathbf{w}$ and $\mathbf{x}$ is less than $90°$. Updating $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}$ yields

$$\mathbf{w}_{\text{new}}^T \mathbf{x} = (\mathbf{w} - \mathbf{x})^T \mathbf{x} = \mathbf{w}^T \mathbf{x} - \mathbf{x}^T \mathbf{x}$$

Since $\mathbf{x}^T \mathbf{x} > 0$, this results in $\mathbf{w}_{\text{new}}^T \mathbf{x} < \mathbf{w}^T \mathbf{x}$, i.e., $\cos(\alpha_{\text{new}}) < \cos(\alpha)$ and thus $\alpha_{\text{new}} > \alpha$. The update moves the vector $\mathbf{w}$ further from $\mathbf{x}$, pushing the decision boundary away from misclassified negative points.

This geometric intuition explains why the perceptron algorithm always converges for linearly separable data. At each step, it reduces the number of misclassified points by rotating the decision boundary in the correct direction.

Definition 1.1. *Two sets P and N of points in an n-dimensional space are called absolutely linearly separable if there exist $n+1$ real numbers $w_0, w_1, \ldots, w_n$ such that every point $(x_1, x_2, \ldots, x_n) \in P$ satisfies $\sum_{i=1}^{n} w_i x_i > w_0$ and every point $(x_1, x_2, \ldots, x_n) \in N$ satisfies $\sum_{i=1}^{n} w_i x_i < w_0$.*

Theorem 1.1. *If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector $\mathbf{w}_t$ only a finite number of times. In other words, if the vectors in P and N are tested cyclically, a weight vector $\mathbf{w}_t$ is found after a finite number of steps t which separates the two sets.*

Proof. If $\mathbf{x} \in N$, then $-\mathbf{x} \in P'$, where $P' = P \cup \{-\mathbf{x} : \mathbf{x} \in N\}$. This is because if $\mathbf{w}^T \mathbf{x} < 0$, then $\mathbf{w}^T(-\mathbf{x}) > 0$.

Hence, we can consider a unified set P' where every element $\mathbf{p} \in P'$ satisfies $\mathbf{w}^T\mathbf{p} \geq 0$. Without loss of generality, normalize all vectors $\mathbf{p}$ so that $\|\mathbf{p}\| = 1$. This normalization does not affect the separating condition since $\mathbf{w}^T\mathbf{p}/\|\mathbf{p}\| \geq 0$ implies $\mathbf{w}^T\mathbf{p} \geq 0$.

Let $\mathbf{w}^*$ be the normalized ideal weight vector such that $\mathbf{w}^{*T}\mathbf{p} > 0$ for all $\mathbf{p} \in P'$. Define $\delta = \min_{\mathbf{p}\in P'} \mathbf{w}^{*T}\mathbf{p} > 0$.

Now, suppose at time t we inspect point $\mathbf{p}_i \in P'$ and find $\mathbf{w}_t^T\mathbf{p}_i \leq 0$. A correction is made:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{p}_i.$$

Let β be the angle between $\mathbf{w}_{t+1}$ and $\mathbf{w}^*$. Then,

$$\cos\beta = \frac{\mathbf{w}^* \cdot \mathbf{w}_{t+1}}{\|\mathbf{w}_{t+1}\|} = \frac{\mathbf{w}^* \cdot (\mathbf{w}_t + \mathbf{p}_i)}{\|\mathbf{w}_{t+1}\|} = \frac{\mathbf{w}^* \cdot \mathbf{w}_t + \mathbf{w}^* \cdot \mathbf{p}_i}{\|\mathbf{w}_{t+1}\|}.$$

Since $\mathbf{w}^* \cdot \mathbf{p}_i \geq \delta$, we obtain

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} \geq \mathbf{w}^* \cdot \mathbf{w}_t + \delta.$$

By induction, after k corrections, we have

$$\mathbf{w}^* \cdot \mathbf{w}_t \geq \mathbf{w}^* \cdot \mathbf{w}_0 + k\delta.$$

Now consider the squared norm of $\mathbf{w}_{t+1}$:

$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t + \mathbf{p}_i\|^2 = \|\mathbf{w}_t\|^2 + 2\mathbf{w}_t \cdot \mathbf{p}_i + \|\mathbf{p}_i\|^2.$$

Since $\mathbf{w}_t \cdot \mathbf{p}_i \leq 0$ and $\|\mathbf{p}_i\| = 1$, we get

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + 1.$$

Inductively,

$$\|\mathbf{w}_t\|^2 \leq \|\mathbf{w}_0\|^2 + k.$$

Putting everything together,

$$\cos\beta = \frac{\mathbf{w}^* \cdot \mathbf{w}_t}{\|\mathbf{w}_t\|} \geq \frac{\mathbf{w}^* \cdot \mathbf{w}_0 + k\delta}{\sqrt{\|\mathbf{w}_0\|^2 + k}}.$$

As k increases, the numerator grows linearly, but the denominator grows as $\sqrt{k}$, so $\cos\beta$ can grow unbounded. However, since $\cos\beta \leq 1$, the number of corrections k must be bounded. Therefore, the perceptron algorithm must converge after a finite number of updates.

Let's have a look back at the questions raised after McCulloch-Pitts model.

What if the inputs are not binary but real-valued? The perceptron works directly with real-valued inputs, using the sign of the weighted sum $\mathbf{w}^\top\mathbf{x}$ to make decisions.

Do we always need to manually specify the threshold parameter? No, the threshold can be learned as a bias term w_0 by appending a constant $x_0 = 1$ to the input vector.

Are all inputs equally important, or should some be assigned more weight? Inputs can have different importance, reflected in the learned weights $\mathbf{w}$, which adjust based on the training data.

Can such a neuron model functions that are not linearly separable? No, a single perceptron can only separate linearly separable data. We will see soon how to handle this.

This section was developed in 1969 by two prominent MIT scientists, ***Marvin Minsky*** and ***Seymour Papert***. In their book, *Perceptrons*, they critically analyzed the limitations of Rosenblatt's model. They mathematically proved the above mentioned *perceptron learning algorithm* and showed that a single perceptron could not solve non-linearly separable problems, such as the simple XOR function.

Their critique wasn't wrong but the fallout was dramatic. Funding dried up. Neural networks were declared a dead end. For nearly a decade, research interest in neural models collapsed.

1.2.2 Layer of Perceptrons

How many boolean functions can you design from n inputs? 2^{2^n}. Some of them are not linear separable. There is no general formula that tells us that. For 2 input example, there are 16 possible boolean functions, of which XOR and !XOR are not linearly separable.

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 1.2: Functions f_1 to f_{16} for input combinations of x_1 and x_2

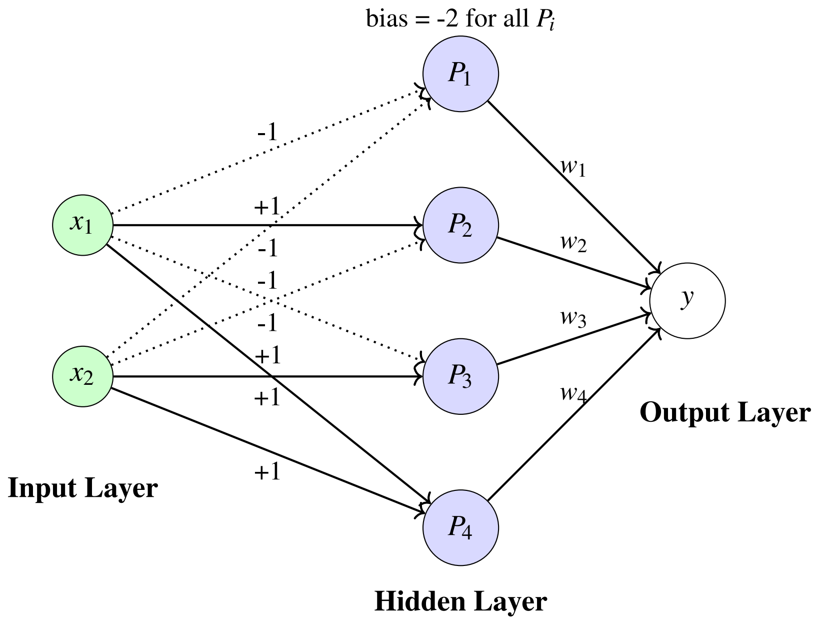


Figure 1.2: Two Input Network with Four Hidden Perceptrons

Two binary inputs, x_1 and x_2, are passed to a hidden layer with four perceptrons P_1 to P_4. Each perceptron in the hidden layer computes a distinct linear combination of the inputs with weights ± 1, effectively separating different input regions. All hidden units use a fixed bias of -2. The outputs from the hidden layer are linearly combined in the output layer using learnable weights w_1, w_2, w_3, w_4 to produce the final output y.

This network can implement *any boolean function*, whether linearly separable or not. The key idea lies in the design of the hidden layer. Each of the four perceptrons is constructed to *fire (output 1) for exactly one of the four possible input combinations* of (x_1, x_2), and *only* for that combination.

Perceptron	**Input that Activates It**
P_1	$(0,0)$
P_2	$(0,1)$
P_3	$(1,0)$
P_4	$(1,1)$

Table 1.3: Unique input activation for each perceptron

Because each hidden unit uniquely represents one input, the output neuron can now learn the desired boolean function by *assigning appropriate weights* w_1, w_2, w_3, w_4 to these hidden activations. For instance, to implement XOR, we can simply set the weights such that the output neuron fires for $(0,1)$ and $(1,0)$, and stays off for $(0,0)$ and $(1,1)$. This translates to the following conditions.

$$w_1 < w_0, \quad w_2 \geq w_0, \quad w_3 \geq w_0, \quad w_4 < w_0$$

There are *no contradictions*. Thus, by adjusting the output weights accordingly, this single network architecture can represent *all 16 possible boolean functions*.

Theorem 1.2. *Any boolean function of n inputs can be represented exactly by a network of perceptrons containing one hidden layer with* 2^n *perceptrons and one output layer containing a single perceptron.*

Proof. A boolean function $f : \{0,1\}^n \to \{0,1\}$ is defined on 2^n possible input vectors.

Construct a hidden layer with 2^n perceptrons, each configured to activate only for one unique input vector by appropriate weights and bias.

The output perceptron then sums the hidden layer outputs with weights chosen to output 1 exactly for inputs where f is 1, and 0 otherwise.

Thus, the network exactly represents any boolean function. □

Note: A network with $2^n + 1$ perceptrons is not necessary but sufficient. For example, we have seen how to represent the OR function (and similarly AND function) with just one perceptron.

Catch: As n increases, the number of perceptrons in the hidden layer increases exponentially.

1.3 The Sigmoid Neuron

The perceptron's thresholding logic is quite harsh. Consider a simple example where we decide whether to like or dislike a movie based on a single input: *the critic's rating* x_1, which ranges from 0 to 1. Suppose the threshold is set at 0.5, with weights $w_0 = -0.5$ and $w_1 = 1$. For a movie with $x_1 = 0.51$, the perceptron outputs *like,* while for $x_1 = 0.49$, it outputs *dislike.* This sudden change in decision seems strict and abrupt.

This behavior is not due to the specific problem or chosen weights. Instead, it is inherent to the perceptron function, which acts as a step function. The output switches sharply from 0 to 1 when the weighted sum $\sum_{i=1}^{n} w_i x_i$ crosses the threshold $-w_0$.

In real-world cases, we usually want a smoother decision function that changes gradually from 0 to 1. This motivates the introduction of sigmoid neurons, where the output function is continuous and smooth.

One common sigmoid function is the logistic function, defined as

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}}$$

Here, the output y does not jump suddenly but transitions smoothly around the threshold $-w_0$.

Moreover, the output y is no longer binary; it takes values between 0 and 1. This output can be interpreted as a probability. So instead of a hard like/dislike decision, we get the probability of liking the movie.

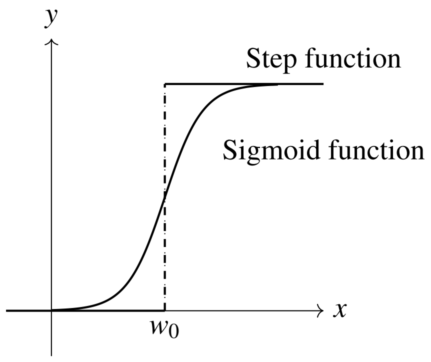


Figure 1.3: Perceptron and Sigmoid Neuron Output Characteristics.

1.3.1 A Typical Supervised Machine Learning Setup

We know that a typical supervised machine learning setup has the following components.

- **Data:** A dataset of n examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, where $\mathbf{x}_i$ are inputs and y_i are corresponding outputs.
- **Model:** An approximation of the relation between input $\mathbf{x}$ and output y. For example,

$$\hat{y} = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}, \quad \hat{y} = \mathbf{w}^\top \mathbf{x}, \quad \hat{y} = \mathbf{x}^\top \mathbf{W} \mathbf{x}$$

  or any other function mapping inputs to outputs.

- **Parameters:** The model depends on parameters $\mathbf{w}$ (and possibly bias b) that need to be learned from the data.
- **Learning Algorithm:** A method to adjust parameters $\mathbf{w}$ and b to fit the data. Examples include perceptron learning or gradient descent (we'll see gradient descent in detail in the next section).
- **Objective/Loss Function:** A function that measures the error of the model predictions. The learning algorithm aims to minimize this loss.

Consider data points (x, y) where $x, y \in \mathbb{R}$.

$$\{(1,\ 0.05),\ (3,\ 0.15),\ (5,\ 0.4),\ (6,\ 0.6),\ (7,\ 0.65),\ (9,\ 0.85),\ (10,\ 0.95)\}$$

Our model is

$$\hat{y} = \frac{1}{1 + e^{-(wx+b)}}$$

We choose Mean Squared Error (MSE) as the loss function.

$$\mathscr{L}(w, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

This loss $\mathscr{L}(w, b)$ defines a surface over the two-dimensional parameter space (w, b). Our goal is to find the (w, b) that minimizes $\mathscr{L}$, the lowest point on this error surface.

Figure 1.4 shows a 3D plot of the MSE loss surface as a function of w and b for the example data.

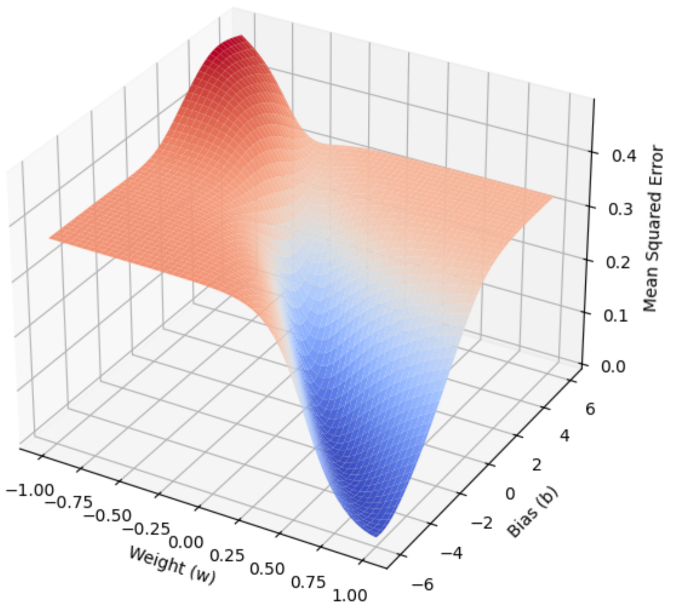


Figure 1.4: 3D Error Surface Over Example Data

1.3.2 Sigmoid Neuron Learning Algorithm: Gradient Descent

Let $\theta = [w, b]$ be the vector of parameters, initialized randomly. Suppose we move in the direction of a change $\Delta\theta = [\Delta w, \Delta b]$. Instead of making a full step, we scale the movement by a small scalar η to stay conservative.

$$\theta_{\text{new}} = \theta + \eta \cdot \Delta\theta$$

What is the right $\Delta\theta$ to choose? To answer this, we turn to the Taylor series expansion. Let $u = \Delta\theta$. Then the Taylor expansion of the loss function $\mathscr{L}(\theta)$ around θ is

$$\mathscr{L}(\theta + \eta u) = \mathscr{L}(\theta) + \eta \cdot u^T \nabla_\theta \mathscr{L}(\theta) + \frac{\eta^2}{2!} \cdot u^T \nabla^2 \mathscr{L}(\theta) u + \frac{\eta^3}{3!} \cdot \ldots$$

Since η is small, the higher-order terms are negligible.

$$\mathscr{L}(\theta + \eta u) \approx \mathscr{L}(\theta) + \eta \cdot u^T \nabla_\theta \mathscr{L}(\theta)$$

For the move to reduce the loss, we require

$$\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) < 0 \quad \Rightarrow \quad u^T \nabla_\theta \mathscr{L}(\theta) < 0$$

To understand how negative this term can be, consider the angle β between u and $\nabla_\theta \mathscr{L}(\theta)$.

$$\cos(\beta) = \frac{u^T \nabla_\theta \mathscr{L}(\theta)}{\|u\| \cdot \|\nabla_\theta \mathscr{L}(\theta)\|} \text{ and } -1 \le \cos(\beta) \le 1$$

Let $k = \|u\| \cdot \|\nabla_\theta \mathscr{L}(\theta)\|$, then

$$-k \le u^T \nabla_\theta \mathscr{L}(\theta) \le k$$

This implies that the most negative value is attained when $\cos(\beta) = -1$, i.e., when the angle is 180°.

$$u = -\nabla_\theta \mathscr{L}(\theta)$$

Hence, the *gradient descent rule* says that the direction u to move in should be opposite to the gradient of the loss.

$$w_{t+1} = w_t - \eta \left. \frac{\partial \mathscr{L}(w,b)}{\partial w} \right|_{w=w_t, b=b_t} \quad \text{and} \quad b_{t+1} = b_t - \eta \left. \frac{\partial \mathscr{L}(w,b)}{\partial b} \right|_{w=w_t, b=b_t}$$

Gradient Descent Algorithm

```
$t \leftarrow 0$
max iterations $\leftarrow 1000$
while $t <$ max iterations do
    $w_{t+1} \leftarrow w_t - \eta \nabla_w \mathscr{L}(w_t, b_t)$
    $b_{t+1} \leftarrow b_t - \eta \nabla_b \mathscr{L}(w_t, b_t)$
    $t \leftarrow t + 1$
end while
```

Starting from initial values $w = 0$ and $b = 0$, gradient descent converges to final parameters $w = 0.1487$, $b = -0.4139$ with a final loss of 0.0527 after 100 iterations.

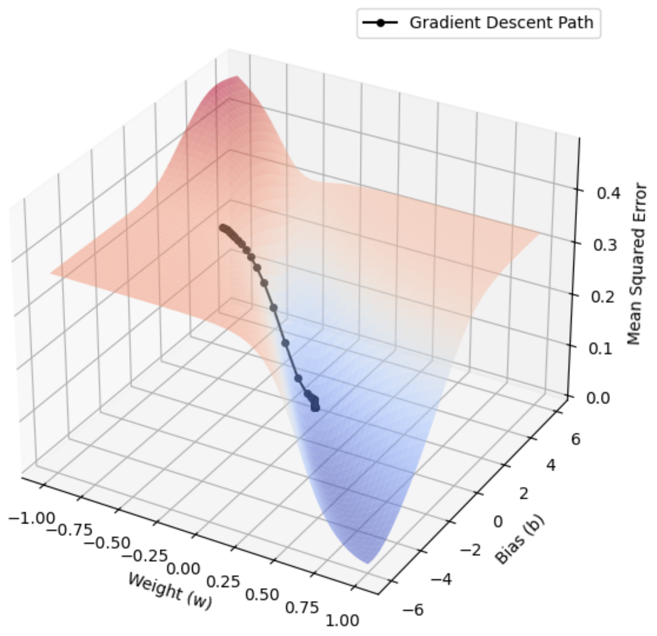


Figure 1.5: Error Surface With Gradient Descent Trajectory.

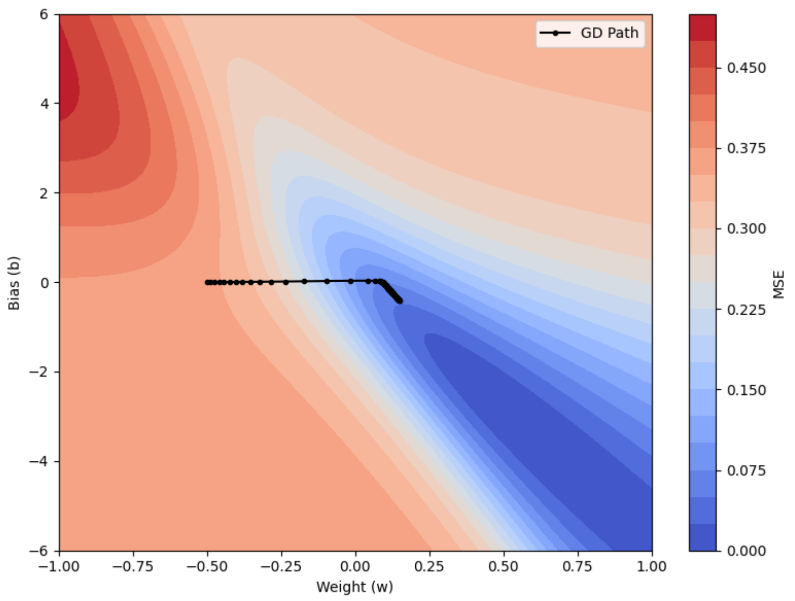


Figure 1.6: Contour Plot of Gradient Descent Trajectory on Error Surface.

Visualizing in three dimensions can get cumbersome. So, we use 2D contour plots instead. When the contours are close together, the slope is steep in that direction.

1.3.3 Momentum Based Gradient Descent

Gradient descent can be slow in regions with gentle slopes. This happens because the gradient becomes very small, leading to small updates. To improve this, we can look at how updates behave over time.

The intuition is that if we keep getting asked to move in the same direction, it's reasonable to trust that direction and take bigger steps — just like a ball gathers speed when rolling downhill.

$$\text{update}_t = \gamma \cdot \text{update}_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - \text{update}_t$$

Here, γ is the momentum coefficient (typically between 0.5 and 0.9), and η is the learning rate. This rule incorporates both the current gradient and the accumulated history of past gradients.

$$\begin{aligned}
\text{update}_0 &= 0 \\
\text{update}_1 &= \gamma \cdot \text{update}_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
\text{update}_2 &= \gamma \cdot \text{update}_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
\text{update}_3 &= \gamma \cdot \text{update}_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
\text{update}_4 &= \gamma \cdot \text{update}_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4
\end{aligned}$$

Continuing this process, we can write the general form,

$$\text{update}_t = \sum_{k=1}^{t} \gamma^{t-k} \cdot \eta \nabla w_k$$

This shows that updates are a weighted sum of all past gradients, with exponentially decaying weights controlled by γ.

Momentum-based Gradient Descent Algorithm

```
$t \leftarrow 0$
max iterations $\leftarrow 1000$
initialize updates: $u_0^w \leftarrow 0, \quad u_0^b \leftarrow 0$
momentum coefficient $\gamma \in [0,1)$
learning rate $\eta > 0$
**while** $t <$ max iterations **do**
    $u_{t+1}^w \leftarrow \gamma \cdot u_t^w + \eta \nabla_w \mathscr{L}(w_t, b_t)$
    $u_{t+1}^b \leftarrow \gamma \cdot u_t^b + \eta \nabla_b \mathscr{L}(w_t, b_t)$
    $w_{t+1} \leftarrow w_t - u_{t+1}^w$
    $b_{t+1} \leftarrow b_t - u_{t+1}^b$
    $t \leftarrow t + 1$
**end while**
```

In our example, consider the initial values $w = 0$ and $b = 0$ and $\gamma = 0.9$. Momentum gradient descent converges to final parameters $w = 0.4328$, $b = -2.3303$ with a final loss of 0.004201 after 100 iterations. This surely performs better than the vanilla gradient descent.

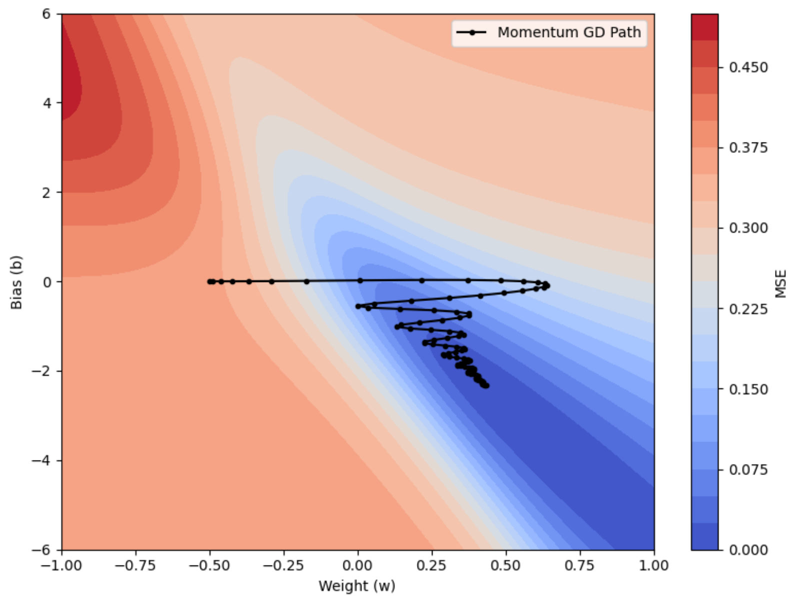


Figure 1.7: Contour Plot of Momentum Gradient Descent Trajectory on Error Surface.

1.3.4 Nesterov Accelerated Gradient Descent

Even in regions with gentle slopes, momentum-based gradient descent takes large steps because momentum carries it forward. It oscillates in and out of the minima valley, often taking many U-turns before finally converging. Despite these oscillations, it converges faster than vanilla gradient descent. But *can we reduce these oscillations?*

The intuition is to *look before we leap*. Recall the momentum update equation,

$$\text{update}_t = \gamma \cdot \text{update}_{t-1} + \eta \nabla_w \mathbf{w}_t$$

We move at least by $\gamma \cdot \text{update}_{t-1}$, plus a bit more by $\eta \nabla_w \mathbf{w}_t$.

What if we calculate the gradient at a look-ahead point instead of the current position? Let's say we define

$$\mathbf{w}_{\text{look ahead}} = \mathbf{w}_t - \gamma \cdot \text{update}_{t-1}$$

Then calculate the gradient $\nabla_w \mathbf{w}_{\text{look ahead}}$ instead of $\nabla_w \mathbf{w}_t$.

Thus, the update rule for Nesterov Accelerated Gradient (NAG) becomes

$$\begin{aligned} \mathbf{w}_{\text{look ahead}} &= \mathbf{w}_t - \gamma \cdot \text{update}_{t-1} \\ \text{update}_t &= \gamma \cdot \text{update}_{t-1} + \eta \nabla_w \mathbf{w}_{\text{look ahead}} \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \text{update}_t \end{aligned}$$

A similar update applies for the bias term b_t.

Nesterov Accelerated Gradient (NAG) Algorithm

```
$t \leftarrow 0$
max iterations $\leftarrow 1000$
initialize $\text{update}^{w}_{-1} \leftarrow \mathbf{0}$, $\text{update}^{b}_{-1} \leftarrow 0$
momentum coefficient $\gamma \in [0,1)$
learning rate $\eta > 0$

while $t <$ max iterations do
    $\mathbf{w}_{\text{look ahead}} \leftarrow \mathbf{w}_t - \gamma \cdot \text{update}^{w}_{t-1}$
    $b_{\text{look ahead}} \leftarrow b_t - \gamma \cdot \text{update}^{b}_{t-1}$

    $\text{update}^{w}_{t} \leftarrow \gamma \cdot \text{update}^{w}_{t-1} + \eta \nabla_w \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$
    $\text{update}^{b}_{t} \leftarrow \gamma \cdot \text{update}^{b}_{t-1} + \eta \nabla_b \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$

    $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \text{update}^{w}_{t}$
    $b_{t+1} \leftarrow b_t - \text{update}^{b}_{t}$
    $t \leftarrow t + 1$
end while
```

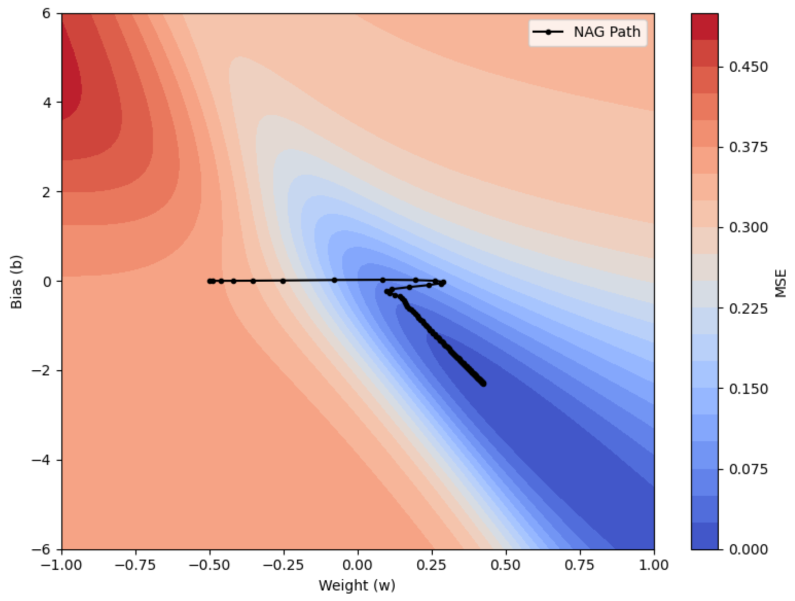


Figure 1.8: Contour Plot of Nesterov Gradient Descent Trajectory on Error Surface.

Looking ahead allows NAG to correct its course faster than momentum-based gradient descent. This reduces oscillations and lowers the chance of escaping the minima valley.

Starting from initial values $w = 0$ and $b = 0$, Nesterov Gradient Descent converges to final parameters $w = 0.4247$, $b = -2.2946$ with a final loss of 0.004477 after 100 iterations.

1.3.5 Stochastic Gradient Descent

Batch gradient descent computes the exact gradient of the loss by averaging the gradients over the entire dataset before updating the parameters. Because the full dataset is used, each update guarantees a decrease in the loss.

However, this method becomes computationally expensive for large datasets. For example, with one million data points, one parameter update requires calculating one million gradients.

Stochastic Gradient Descent (SGD) improves efficiency by updating parameters after evaluating the gradient on each individual data point. Hence, for one million data points, SGD performs one million updates per epoch, where an *epoch* is defined as a full pass over the dataset.

However, the gradient used in SGD is a noisy, stochastic estimate of the true gradient since it is computed using only one data point rather than the entire dataset. Due to this noise, there is no guarantee that each SGD update reduces the overall loss function. Indeed, the updates tend to oscillate, especially in small datasets, as each data point tries to optimize locally without considering the global effect on other points.

Even with these jumps, SGD can still find a good solution in the long run, as long as the learning rate is adjusted properly over time.

To smooth things out, we can use mini-batches. Instead of just one point, we use a small group of data points to compute the gradient. This reduces the noise while keeping the updates fast. It also helps the model converge more smoothly.

Also, the randomness in SGD isn't always bad. It can help the model escape flat spots or shallow traps in the loss surface, which sometimes leads to better results.

Mini-Batch Stochastic Gradient Descent Algorithm

```
$t \leftarrow 0$
max iterations $\leftarrow 1000$
mini-batch size $\leftarrow m$

**while** $t <$ max iterations **do**
    Sample mini-batch $\mathscr{B}_t$ of size $m$ from training data

    Compute gradients over mini-batch:
        $g_w \leftarrow \frac{1}{m}\sum_{(x_i,y_i)\in\mathscr{B}_t} \nabla_w \mathscr{L}(w_t, b_t;\, x_i, y_i)$
        $g_b \leftarrow \frac{1}{m}\sum_{(x_i,y_i)\in\mathscr{B}_t} \nabla_b \mathscr{L}(w_t, b_t;\, x_i, y_i)$

        $w_{t+1} \leftarrow w_t - \eta g_w$
        $b_{t+1} \leftarrow b_t - \eta g_b$

    $t \leftarrow t + 1$
**end while**
```

Starting from initial values $w = 0$ and $b = 0$, Stochastic Gradient Descent converges to final parameters $w = 0.3706$, $b = -1.9689$ with a final loss of 0.007861 after 100 iterations.

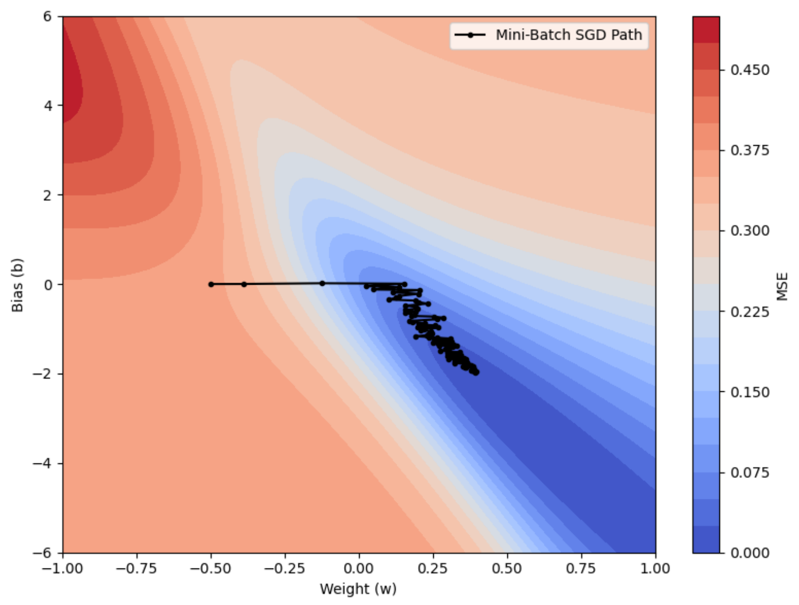


Figure 1.9: Contour Plot of Stochastic Gradient Descent Trajectory on Error Surface.

1.3.6 Gradient Descent with Adaptive Learning

One might think to solve the problem of navigating gentle slopes by setting a high learning rate η, effectively amplifying small gradients. But it is better to have a learning rate that adapts to the gradient itself.

Intuition 1 (Adagrad): Decay the learning rate for each parameter based on its past updates. More updates lead to more decay. The update rule for Adagrad is

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \cdot \nabla w_t$$

A similar set of equations applies for the bias parameter b_t.

Batch Gradient Descent with Adagrad

```
$t \leftarrow 0$ , max iterations $\leftarrow 1000$
$v_w \leftarrow 0, v_b \leftarrow 0$
**while** $t <$ max iterations **do**
    $g_{w_t} \leftarrow \nabla_w \mathscr{L}(w_t, b_t)$ ,    $g_{b_t} \leftarrow \nabla_b \mathscr{L}(w_t, b_t)$
    $v_w \leftarrow v_w + g_{w_t}^2$ ,    $v_b \leftarrow v_b + g_{b_t}^2$
    $w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{v_w}+\varepsilon} \cdot g_{w_t}$ ,    $b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{v_b}+\varepsilon} \cdot g_{b_t}$
    $t \leftarrow t + 1$
**end while**
```

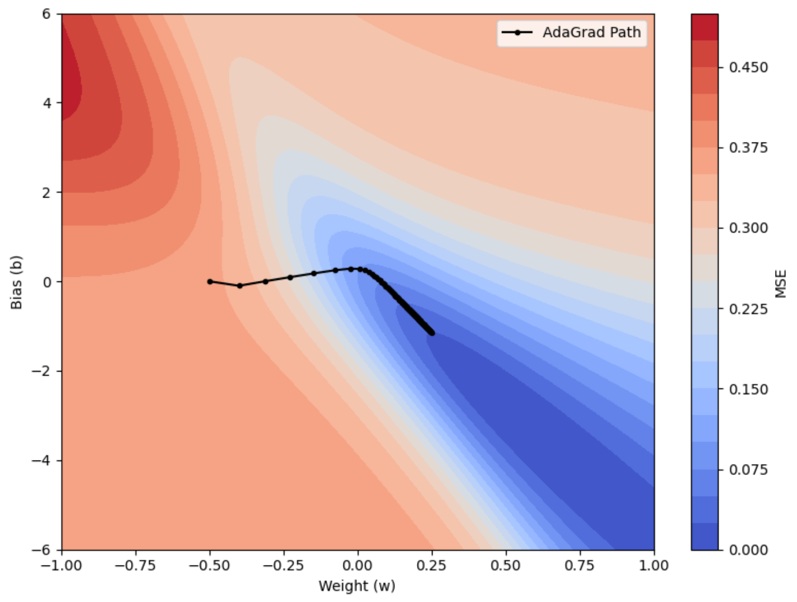


Figure 1.10: Contour Plot of Gradient Descent Trajectory with Adagrad Learning on Error Surface.

Starting from initial values $w = 0$ and $b = 0$, Gradient Descent with Adagrad Learning converges to final parameters $w = 0.2493$, $b = -1.1415$ with a final loss of 0.024577 after 100 iterations.

Intuition 2 (RMSProp): Adagrad decays the learning rate too aggressively. After some time, frequently updated parameters get very small updates because the denominator $\sqrt{v_t}$ grows without bound. To fix this, RMSProp decays v_t to prevent its rapid growth.

The update rule for RMSProp is

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \cdot \nabla w_t$$

Again, a similar set of equations applies for b_t.

Batch Gradient Descent with RMSProp

```
$t \leftarrow 0$ , max iterations $\leftarrow 1000$
Initialize $v_w \leftarrow 0$, $v_b \leftarrow 0$
Choose $\eta, \beta, \varepsilon$
**while** $t <$ max iterations **do**
    $g_w \leftarrow \nabla_w \mathscr{L}(w_t, b_t)$,    $g_b \leftarrow \nabla_b \mathscr{L}(w_t, b_t)$
    $v_w \leftarrow \beta \cdot v_w + (1-\beta) \cdot g_w^2$ ,    $v_b \leftarrow \beta \cdot v_b + (1-\beta) \cdot g_b^2$
    $w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{v_w}+\varepsilon} \cdot g_w$ ,    $b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{v_b}+\varepsilon} \cdot g_b$
    $t \leftarrow t+1$
**end while**
```

Adagrad gets stuck near convergence because its learning rate decays too much, making it hard to move in some directions (like the vertical b direction). RMSProp fixes this by decaying v_t less aggressively.

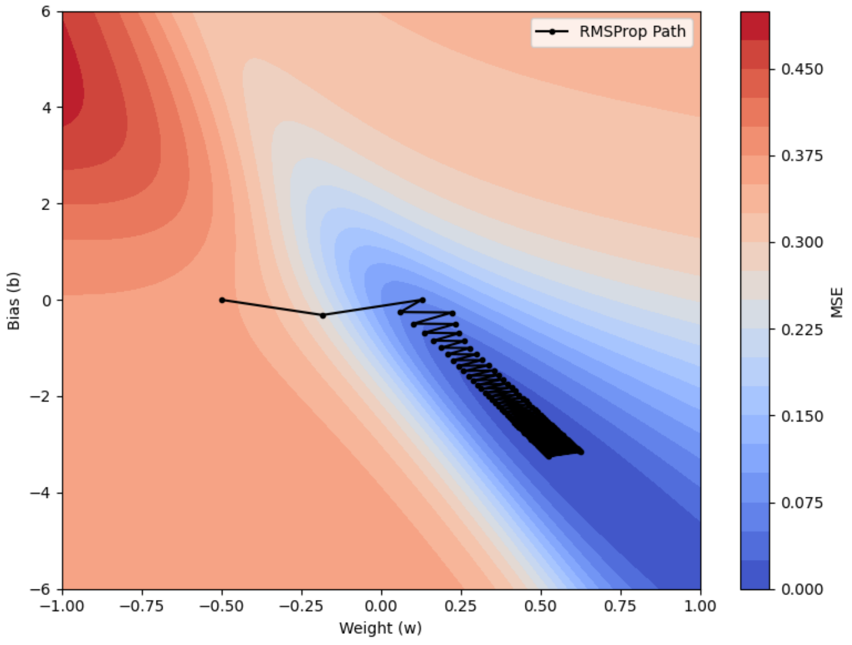


Figure 1.11: Contour Plot of Gradient Descent Trajectory with RMSProp Learning on Error Surface.

Starting from initial values $w = 0$ and $b = 0$, Gradient Descent with RMSProp Learning converges to final parameters $w = 0.6249$, $b = -3.1551$ with a final loss of 0.005095 after 100 iterations.

Intuition 3 (Adam): Adam builds on RMSProp by also keeping a cumulative history of gradients (first moment estimate).

The update rules for Adam are

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla w_t$$

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla w_t)^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t}+\varepsilon} \cdot \hat{m}_t$$

A similar set of equations applies for b_t.

Starting from initial values $w = 0$ and $b = 0$, Gradient Descent with Adam Learning converges to final parameters $w = 0.5227$, $b = -2.9403$ with a final loss of 0.001289 after 100 iterations.

Batch Gradient Descent with Adam

```
$t \leftarrow 0$,    max iterations $\leftarrow 1000$
Initialize $m_w = 0,\ v_w = 0,\ m_b = 0,\ v_b = 0$
Choose $\beta_1, \beta_2$

while $t <$ max iterations do
    Compute gradients: $g_w \leftarrow \nabla_w \mathscr{L}(w_t, b_t)$, $g_b \leftarrow \nabla_b \mathscr{L}(w_t, b_t)$

    $m_w \leftarrow \beta_1 \cdot m_w + (1-\beta_1) \cdot g_w$ ,    $v_w \leftarrow \beta_2 \cdot v_w + (1-\beta_2) \cdot g_w^2$
    $\hat{m}_w \leftarrow \frac{m_w}{1-\beta_1^t}$ ,    $\hat{v}_w \leftarrow \frac{v_w}{1-\beta_2^t}$
    $w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{\hat{v}_w}+\varepsilon} \cdot \hat{m}_w$

    $m_b \leftarrow \beta_1 \cdot m_b + (1-\beta_1) \cdot g_b$ ,    $v_b \leftarrow \beta_2 \cdot v_b + (1-\beta_2) \cdot g_b^2$
    $\hat{m}_b \leftarrow \frac{m_b}{1-\beta_1^t}$ ,    $\hat{v}_b \leftarrow \frac{v_b}{1-\beta_2^t}$
    $b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{\hat{v}_b}+\varepsilon} \cdot \hat{m}_b$

    $t \leftarrow t+1$
end while
```

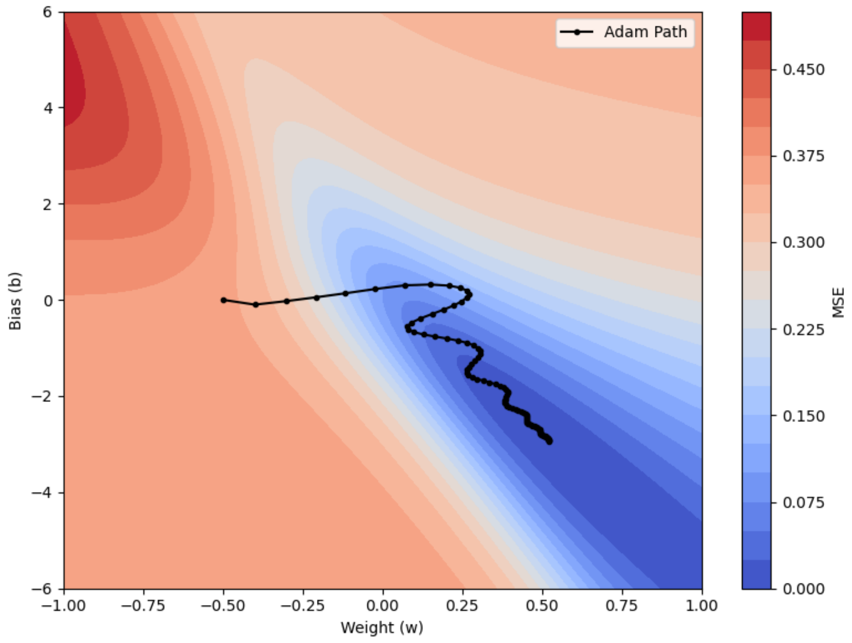


Figure 1.12: Contour Plot of Gradient Descent Trajectory with Adam Learning on Error Surface.

All these adaptive learning rate methods modify the vanilla gradient descent update. Adam can also be combined with Nesterov's lookahead method for further improvements.

Nesterov-Adam (Nadam) Algorithm

```
$t \leftarrow 0,$    max iterations $\leftarrow 1000$
Initialize $m_w = 0,\ v_w = 0,\ m_b = 0,\ v_b = 0$
Initialize $w_0,\ b_0,$    momentum coefficient $\gamma = \beta_1 \in [0,1),$    learning rate $\eta > 0$
while $t <$ max iterations do
    $\mathbf{w}_{\text{look ahead}} \leftarrow \mathbf{w}_t - \gamma \cdot m_w$
    $b_{\text{look ahead}} \leftarrow b_t - \gamma \cdot m_b$
    Compute gradients: $g_w \leftarrow \nabla_w \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}}),$    $g_b \leftarrow \nabla_b \mathscr{L}(\mathbf{w}_{\text{look ahead}}, b_{\text{look ahead}})$

    $m_w \leftarrow \beta_1 \cdot m_w + (1-\beta_1) \cdot g_w\ ,$    $v_w \leftarrow \beta_2 \cdot v_w + (1-\beta_2) \cdot g_w^2$
    $\hat{m}_w \leftarrow \frac{m_w}{1-\beta_1^{t+1}}\ ,$    $\hat{v}_w \leftarrow \frac{v_w}{1-\beta_2^{t+1}}$
    $w_{t+1} \leftarrow w_t - \frac{\eta}{\sqrt{\hat{v}_w}+\varepsilon} \cdot (\gamma \cdot \hat{m}_w + (1-\gamma) \cdot g_w)$

    $m_b \leftarrow \beta_1 \cdot m_b + (1-\beta_1) \cdot g_b\ ,$    $v_b \leftarrow \beta_2 \cdot v_b + (1-\beta_2) \cdot g_b^2$
    $\hat{m}_b \leftarrow \frac{m_b}{1-\beta_1^{t+1}}\ ,$    $\hat{v}_b \leftarrow \frac{v_b}{1-\beta_2^{t+1}}$
    $b_{t+1} \leftarrow b_t - \frac{\eta}{\sqrt{\hat{v}_b}+\varepsilon} \cdot (\gamma \cdot \hat{m}_b + (1-\gamma) \cdot g_b)$
    $t \leftarrow t+1$
end while
```

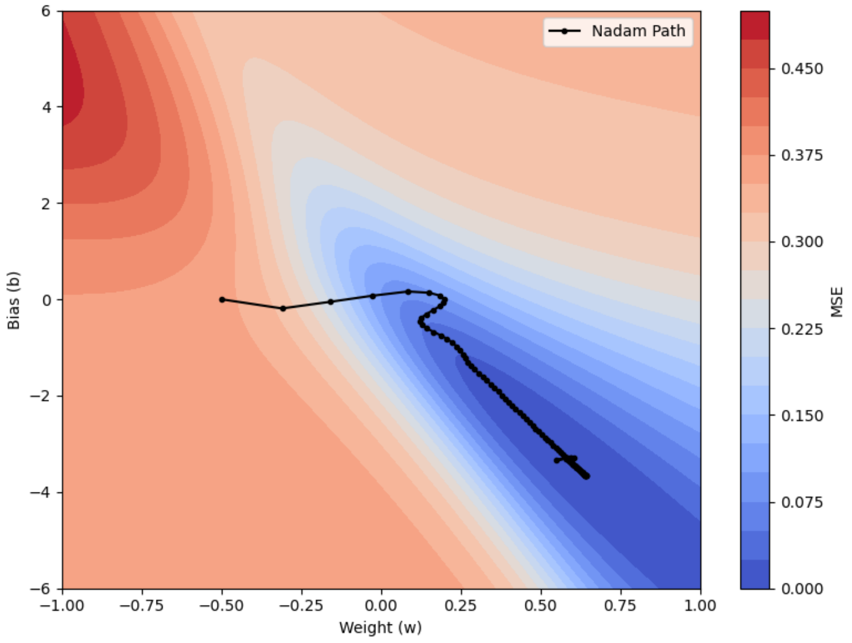


Figure 1.13: Contour Plot of Nesterov Gradient Descent Trajectory with Adam Learning on Error Surface.

Starting from initial values $w = 0$ and $b = 0$, Nesterov Gradient Descent with Adam Learning converges to final parameters $w = 0.5829$, $b = -3.3043$ with a final loss of 0.000833 after 100 iterations.

Method	**Final Parameters (w, b)**	**Final Loss**
Vanilla Gradient Descent	$w = 0.1487,\ b = -0.4139$	0.0527
Momentum Gradient Descent ($\gamma = 0.9$)	$w = 0.4328,\ b = -2.3303$	0.004201
Nesterov Gradient Descent	$w = 0.4247,\ b = -2.2946$	0.004477
Stochastic Gradient Descent	$w = 0.3706,\ b = -1.9689$	0.007861
Adagrad Gradient Descent	$w = 0.2493,\ b = -1.1415$	0.024577
RMSProp Gradient Descent	$w = 0.6249,\ b = -3.1551$	0.005095
Adam Gradient Descent	$w = 0.5227,\ b = -2.9403$	0.001289
Nesterov with Adam (Best)	$w = 0.5829,\ b = -3.3043$	**0.000833**

Table 1.4: Comparison of Gradient Descent Methods (Initial values: $w = 0$, $b = 0$, 100 iterations)

1.3.7 Layer of Sigmoid Neurons

Recall a multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors). Similarly, the representation power of a multilayer network of sigmoid neurons is given by the **Universal Approximation Theorem.** To prove this, we will need the result from **Stone-Weierstrass Theorem**.

Theorem 1.3. ***Stone-Weierstrass Theorem.*** *Let K be a compact subset of $\mathbb{R}^n$ and let $\mathscr{A}$ be a subalgebra of $C(K)$ (the space of continuous real-valued functions on K) such that*

1. *$\mathscr{A}$ separates points of K (i.e., for any $x, y \in K$ with $x \neq y$, there exists $f \in \mathscr{A}$ such that $f(x) \neq f(y)$)*
2. *$\mathscr{A}$ contains the constant functions*

Then $\mathscr{A}$ is dense in $C(K)$ with respect to the uniform norm.

Imagine you have a collection of continuous functions defined on a *nice* and *closed* space (like a closed and bounded interval). If this collection of functions has two key properties:

1. *They can tell points apart.* For any two distinct points in your space, you can find a function in your collection that gives different values for those two points.
2. *They include all the constant functions.* You have functions in your collection that are just a fixed number everywhere.

Then, you can use combinations of these functions (adding them, multiplying them, multiplying them by constants) to get arbitrarily close to any other continuous function on that space.

Essentially, if your initial collection of functions is rich enough to distinguish between points and includes constants, it's *dense* enough to build approximations for any other continuous function you can think of on that space.

Proof. Let $f \in C(K)$ and $\varepsilon > 0$. We aim to show that there exists a function $g \in \mathscr{A}$ such that

$$\|f - g\|_\infty < \varepsilon.$$

Step 1: Approximate indicator functions.

Fix $x_0 \in K$. For any $x \in K$, since $\mathscr{A}$ separates points, there exists $\phi_x \in \mathscr{A}$ such that $\phi_x(x) \neq \phi_x(x_0)$. Define:

$$\psi_x(y) = \left(\frac{\phi_x(y) - \phi_x(x_0)}{\phi_x(x) - \phi_x(x_0)} \right)^2.$$

Then $\psi_x \in \mathscr{A}$ because $\mathscr{A}$ is a subalgebra (closed under addition, multiplication, and scalar multiplication), and we've used only these operations. Also, $\psi_x(x_0) = 0$, and $\psi_x(x) = 1$.

Now, for a finite set $\{x_1, \dots, x_m\} \subset K$, we can construct a partition of unity-like approximation by defining functions $\{\psi_{x_j}\}_{j=1}^m$ and then normalize

$$S(y) = \sum_{j=1}^m \psi_{x_j}(y), \quad \rho_j(y) = \frac{\psi_{x_j}(y)}{S(y)}.$$

Each $\rho_j \in \mathscr{A}$, and $\sum_{j=1}^m \rho_j(y) = 1$ for all $y \in K$.

Step 2: Local approximation of f.

Since f is uniformly continuous on compact K, there exists a finite δ-net $\{x_1, \dots, x_m\} \subset K$ such that for all $y \in K$, there exists x_j with $|f(y) - f(x_j)| < \varepsilon$.

Now define the function

$$g(y) = \sum_{j=1}^m f(x_j)\rho_j(y).$$

Then $g \in \mathscr{A}$ because $\rho_j \in \mathscr{A}$ and $f(x_j)$ are constants (constants are in $\mathscr{A}$, and the algebra is closed under scalar multiplication and addition).

Step 3: Approximation bound.

We estimate the uniform difference

$$|f(y) - g(y)| = \left| f(y) - \sum_{j=1}^m f(x_j)\rho_j(y) \right| = \left| \sum_{j=1}^m \rho_j(y)(f(y) - f(x_j)) \right|.$$

Using the triangle inequality

$$|f(y) - g(y)| \leq \sum_{j=1}^m \rho_j(y)|f(y) - f(x_j)|.$$

Since each $\rho_j(y) \geq 0$, and $\sum \rho_j(y) = 1$, this is a convex combination of errors $|f(y) - f(x_j)|$, each less than ε. So,

$$|f(y) - g(y)| < \varepsilon \quad \text{for all } y \in K.$$

Hence,

$$\|f - g\|_\infty < \varepsilon.$$

□

Example 1.1. *Polynomials can be used to approximate any continuous function on a closed interval.*

Solution 1.1. Let K be a compact subset of $\mathbb{R}^n$. We consider the subalgebra $\mathscr{A}$ of $C(K)$ consisting of all polynomial functions on K. For clarity, let's first consider the common case where K is a closed and bounded interval in $\mathbb{R}$, say $K = [a,b]$.

A function $f \in \mathscr{A}$ has the form $f(x) = c_m x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0$ for some non-negative integer m and real coefficients $c_0, c_1, \ldots, c_m$.

We need to demonstrate that $\mathscr{A}$ satisfies the two properties required by the Stone-Weierstrass Theorem.

1. $\mathscr{A}$ separates points of K

This property requires that for any distinct points $x, y \in K$ (i.e., $x \neq y$), there must exist a function $f \in \mathscr{A}$ such that $f(x) \neq f(y)$.

Consider any two distinct points $x, y \in [a,b]$ such that $x \neq y$. Let's choose the polynomial function $f(t) = t$. This is a simple polynomial of degree 1 (where $m = 1$, $c_1 = 1$, and $c_0 = 0$). Evaluating this function at x and y, we get $f(x) = x$ and $f(y) = y$. Since we initially assumed $x \neq y$, it directly follows that $f(x) \neq f(y)$.

Therefore, the set of polynomial functions $\mathscr{A}$ successfully separates points of K.

2. $\mathscr{A}$ contains the constant functions

This property requires that for any real number c, the constant function $g(t) = c$ for all $t \in K$ must be an element of $\mathscr{A}$.

A constant function $g(t) = c$ can be expressed as a polynomial of degree 0. We can write $g(t) = c \cdot t^0$, or simply $g(t) = c$. This fits the general form of a polynomial where $m = 0$ and $c_0 = c$.

Thus, for any real number c, the constant function $g(t) = c$ is indeed a polynomial.

Therefore, the set of polynomial functions $\mathscr{A}$ contains all constant functions.

Since the set of polynomial functions on $K = [a,b]$ satisfies both of these crucial properties, the Stone-Weierstrass Theorem implies that the set of polynomials is **dense** in $C([a,b])$.

This is a fundamental result, meaning that any continuous real-valued function defined on a closed and bounded interval can be uniformly approximated arbitrarily well by polynomials. The argument extends naturally to compact subsets of $\mathbb{R}^n$. In this case, $\mathscr{A}$ would comprise polynomials in n variables.

Refer figure 1.14 for illustration.

Theorem 1.4. ***Universal Approximation Theorem.*** *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded, and continuous function (such as the sigmoid function). Let $K \subset \mathbb{R}^n$ be compact. Then for any continuous function $f : K \to \mathbb{R}$ and any $\varepsilon > 0$, there exist $N \in \mathbb{N}$, weights $\alpha_1, \ldots, \alpha_N \in \mathbb{R}$, weight vectors $\mathbf{w}_1, \ldots, \mathbf{w}_N \in \mathbb{R}^n$, and biases $b_1, \ldots, b_N \in \mathbb{R}$ such that the function*

$$g(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

satisfies $\sup_{\mathbf{x} \in K} |f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon$.

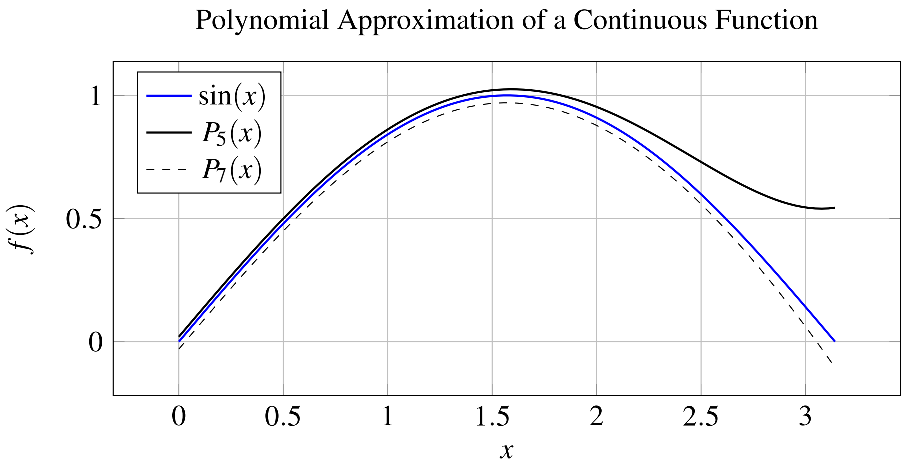


Figure 1.14: Illustration of the Stone-Weierstrass theorem. Polynomials $P_5(x)$ and $P_7(x)$ successively approximate $\sin(x)$ with increasing accuracy on the compact interval $[0,\pi]$.

Proof. We want to show that the set of functions

$$\mathscr{F} = \left\{ \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j) : N \in \mathbb{N},\ \alpha_j \in \mathbb{R},\ \mathbf{w}_j \in \mathbb{R}^n,\ b_j \in \mathbb{R} \right\}$$

is dense in $C(K)$, the space of continuous real-valued functions on compact $K \subset \mathbb{R}^n$, with respect to the uniform norm. Let us define

$$\mathscr{A} = \operatorname{span}\{\sigma(\mathbf{w}^T \mathbf{x} + b) : \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R}\},$$

which is a subalgebra of $C(K)$.

Step 1: Closure under addition, multiplication, and scalar multiplication.

The set $\mathscr{A}$ is a linear span of compositions of affine functions with σ, so it is closed under addition and scalar multiplication by construction.

Step 2: Separation of points.

Let $\mathbf{x}, \mathbf{y} \in K$ with $\mathbf{x} \neq \mathbf{y}$. Since σ is non-constant and continuous, there exists a hyperplane (i.e., choice of $\mathbf{w} \in \mathbb{R}^n$) such that $\mathbf{w}^T \mathbf{x} \neq \mathbf{w}^T \mathbf{y}$. Then for any fixed b, we have

$$\sigma(\mathbf{w}^T \mathbf{x} + b) \neq \sigma(\mathbf{w}^T \mathbf{y} + b),$$

because σ is strictly increasing (as in sigmoid), or at least non-constant and continuous, so it distinguishes between different inputs. Hence, $\mathscr{A}$ separates points of K.

Step 3: Constants are in $\mathscr{A}$.

Since σ is bounded and non-constant, it attains at least two values. For large positive or negative arguments, $\sigma(z) \to c_1$ or c_2, so we can scale and shift σ to approximate constant functions arbitrarily well. More directly, linear combinations of shifted sigmoids can approximate any constant function. Hence, $\mathscr{A}$ contains constant functions or can approximate them arbitrarily well, which is enough for density in the uniform norm.

By the Stone-Weierstrass Theorem 1.3, $\mathscr{A}$ is dense in $C(K)$. Thus, for any $f \in C(K)$ and $\varepsilon > 0$, there exists a finite sum of the form

$$g(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$$

such that

$$\sup_{\mathbf{x} \in K} |f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon.$$

□

Corollary 1.1. *A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.*

In other words, there is a guarantee that for any function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$, we can always find a neural network (with one hidden layer containing enough sigmoid neurons) whose output $g(\mathbf{x})$ satisfies $|g(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$.

1.4 Neural Network

1.4.1 A Typical Supervised Machine Learning Setup

1.4.2 Neural Network Learning Algorithm: Backpropagation

1.4.3 Practical: MNIST Digit Classification

Chapter 2

Low-Rank Representations & Autoencoding

Motivation behind how vectors are the fundamental quantity that we deal with. Everything to be processed is eventually a vector.

2.1 Refresher on Linear Algebra

2.1.1 Linear Independence

2.1.2 Orthogonality

2.1.3 Eigenvalues and Eigenvectors

2.1.4 Eigenvalue Decomposition

2.2 Principal Component Analysis

2.2.1 Interpretation 1: Best Low-Rank Representation

2.2.2 Interpretation 2: Minimum Covariance In New Basis

2.2.3 Interpretation 3: High Variance In New Basis

2.3 Singular Value Decomposition

2.4 Introduction to Autoencoders

2.5 Link Between PCA and Autoencoders

2.6 Denoising Autoencoders

2.7 Sparse Autoencoders

2.8 Contractive Autoencoders

Chapter 3

Better Model Training Techniques

Out of two models, which one is better? And what are the tips and techniques that we shall follow in order to make a better model?

3.1 Bias and Variance

3.1.1 The Tradeoff

3.1.2 Train Error vs. Test Error

3.1.3 True Error and Model Complexity

3.2 Regularization

3.2.1 l_2 Regularization

3.2.2 Dataset Augmentation

3.2.3 Parameter Sharing and Tying

3.2.4 Adding Noise to the Inputs

3.2.5 Adding Noise to the Outputs

3.2.6 Early Stopping

3.2.7 Ensemble Methods

3.2.8 Dropout

3.3 Better Activation Functions

3.4 Better Weight Initialization

3.5 Batch Normalization

SECTION 2

LANGUAGE AND TEXT

Chapter 4

Introduction to Natural Language Processing

Since computers can only understand the numbers, the question was: how to teach computers about the language?

Why? What's the motivation for this? What tasks do we want to do?

4.1 N-Gram Language Model

4.2 Naive Bayes' Model

4.3 Logistic Regression for Text Classification

Chapter 5

Word Embeddings

Some intro

5.1 Word2Vec Model

5.2 Skip Gram Model

5.3 Feedforward Neural Language Model

5.4 Recurrent Neural Network

5.5 Long Short Term Memory Cells

Chapter 6

The Transformer

Some intro

6.1 Attention

6.2 Transformer Blocks

6.3 Computational Enhancements

6.4 Transformer Language Model

Chapter 7

Large Language Models

Some intro

7.1 Transformer Architecture

7.2 Sampling for LLMs

7.3 Pretraining LLMs

7.4 Evaluating LLMs

7.5 Scaling LLMs

7.6 Problems with LLMs

Chapter 8

Masked Language Models

Some intro

8.1 Bidirectional Transformer Encoders

8.2 Training Bidirectional Encoders

8.3 Contextual Embeddings

8.4 Fine-Tuning for Classification

Chapter 9

Prompting and Model Alignment

Some intro

9.1 Prompting

9.2 Post-Training and Model Alignment

9.3 Prompt Engineering

Chapter 10

Retrieval Augmented Generation

Chapter 11

LangChain and LangSmith

Chapter 12

LangGraph and AI Agents

Chapter 13

Ollama Ecosystem

SECTION 3

VISION AND IMAGES

SECTION 4

AUDIO AND SPEECH

SECTION 5

VIDEO, EMBODIMENT, AND INTERACTION

SECTION 6

SYSTEMS, ETHICS, AND THE FUTURE

Bibliography

[1] Warren S. McCulloch; Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics, Society for Mathematical Biology*, 5, 1943.